Artificial Intelligence Lab
AL-2002

Lab 06

Instructor: Hurmat Hidayat

Semester: Spring 2023

Artificial Intelligence Lab 06

Objectives

The objective of this AI lab is to provide students with a comprehensive understanding of supervised machine learning, specifically Multilayer Layer Perceptron.

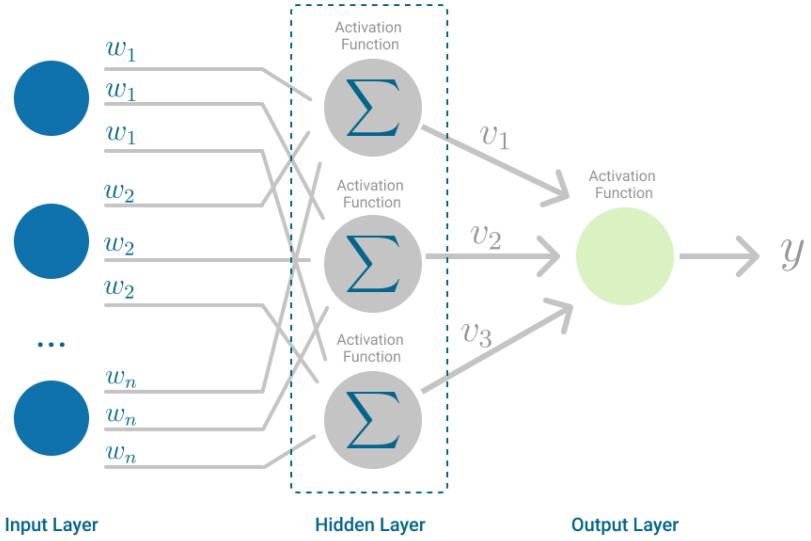
Learning Outcomes

1. Apply the Multilayer Perceptron to real-world datasets and analyze the results.

Table of Contents

Multilayer Perceptron

A Multilayer Perceptron has input and output layers, and one or more **hidden layers** with many neurons stacked together. And while in the Perceptron the neuron must have an activation function that imposes a threshold, like ReLU or sigmoid, neurons in a Multilayer Perceptron can use any arbitrary activation function.

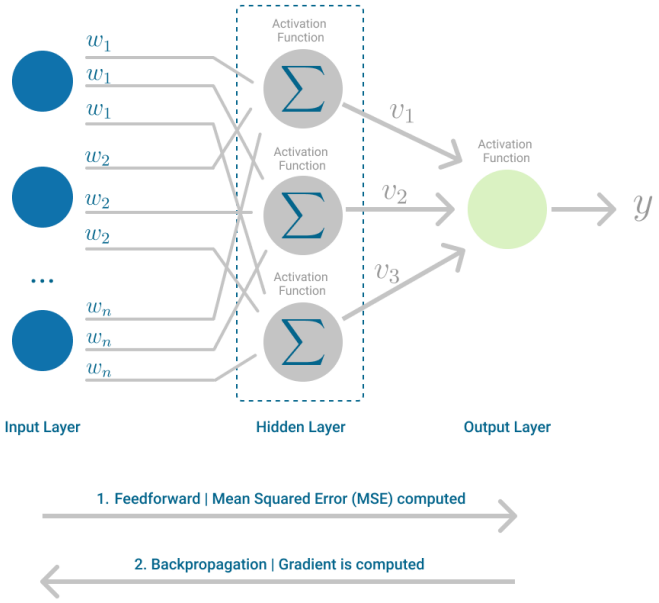


Multilayer Perceptron falls under the category of **feedforward algorithms**, because inputs are combined with the initial weights in a weighted sum and subjected to the activation function, just like in the Perceptron. But the difference is that each linear combination is propagated to the next layer. Each layer is *feeding* the next one with the result of their computation, their internal representation of the data. This goes all the way through the hidden layers to the output layer. If the algorithm only computed the weighted sums in each neuron, propagated results to the output layer, and stopped there, it wouldn't be able to *learn* the weights that minimize the cost function. If the algorithm only computed one iteration, there would be no actual learning. This is where **Backpropagation** comes into play.

Backpropagation

Backpropagation is the learning mechanism that allows the Multilayer Perceptron to iteratively adjust the weights in the network, with the goal of minimizing the cost function.

There is one hard requirement for backpropagation to work properly. The function that combines inputs and weights in a neuron, for instance the weighted sum, and the threshold function, for instance ReLU, must be differentiable. These functions must have a **bounded derivative**, because Gradient Descent is typically the optimization function used in MultiLayer Perceptron.



In each iteration, after the weighted sums are forwarded through all layers, the gradient of the **Mean Squared Error** is computed across all input and output pairs. Then, to propagate it back, the weights of the first hidden layer are updated with the value of the gradient. That's how the weights are propagated back to the starting point of the neural network!

$$\underbrace{\Delta_w(t)}_{\text{Gradient Current Iteration}} = \overbrace{-\varepsilon}^{\text{Bias}} \underbrace{\frac{\overbrace{dE}^{\text{Error}}}{\underbrace{dw_{(t)}}_{\text{Weight vector}}}}_{} + \overbrace{\alpha}^{\text{Learning Rate}} \underbrace{\Delta_{w(t-1)}}_{\text{Gradient Previous Iteration}}$$

This process keeps going until gradient for each input-output pair has converged, meaning the newly computed gradient hasn't changed more than a specified *convergence threshold*, compared to the previous iteration.

How Many Hidden Layers/Neurons to Use in Artificial Neural Networks?

ANN is inspired by the biological neural network. For simplicity, in computer science, it is represented as a set of layers. These layers are categorized into three classes which are input, hidden, and output.

Knowing the number of input and output layers and the number of their neurons is the easiest part. Every network has a single input layer and a single output layer. The number of neurons in the input layer equals the number of input variables in the data being processed. The number of neurons in the output layer equals the number of outputs associated with each input. But the challenge is knowing the number of hidden layers and their neurons.

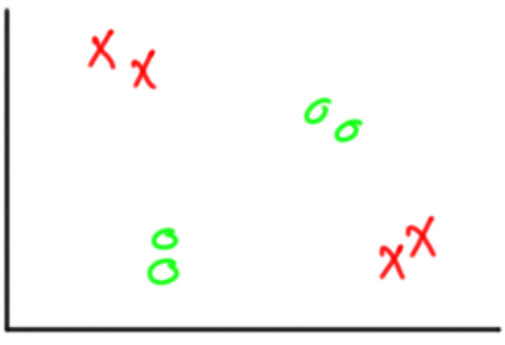
Guidelines

Here are some guidelines to know the number of hidden layers and neurons per each hidden layer in a classification problem:

1. Based on the data, draw an expected decision boundary to separate the classes.
2. Express the decision boundary as a set of lines. Note that the combination of such lines must yield to the decision boundary.
3. The number of selected lines represents the number of hidden neurons in the first hidden layer.
4. To connect the lines created by the previous layer, a new hidden layer is added. Note that a new hidden layer is added each time you need to create connections among the lines in the previous hidden layer.
5. The number of hidden neurons in each new hidden layer equals the number of connections to be made.

Example

Let's start with a simple example of a classification problem with two classes as shown in figure. Each sample has two inputs and one output that represents the class label. It is much similar to XOR problem.



The first question to answer is whether hidden layers are required or not. A rule to follow in order to determine whether hidden layers are required or not is as follows:

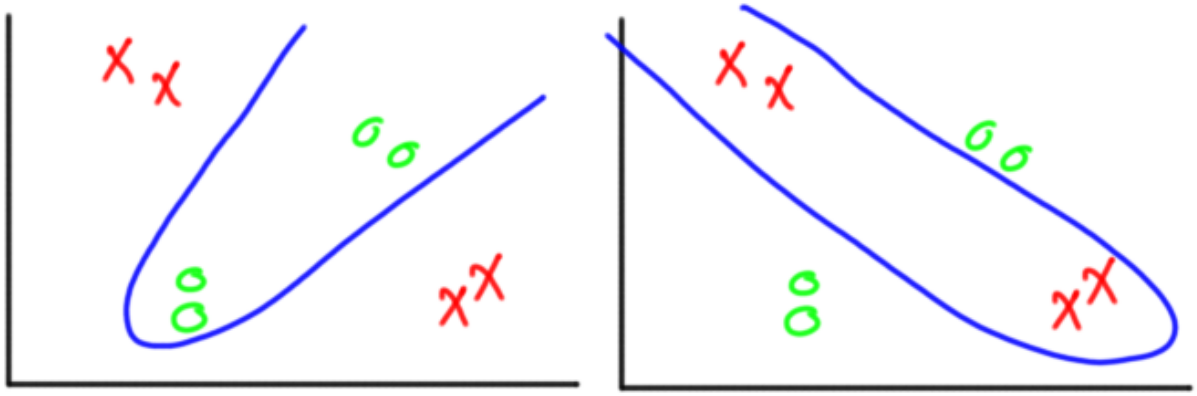
In artificial neural networks, hidden layers are required if and only if the data must be separated non-linearly.

Looking at figure 2, it seems that the classes must be non-linearly separated. A single line will not work. As a result, we must use hidden layers in order to get the best decision boundary. In such case, we may still not use hidden layers but this will affect the classification accuracy. So, it is better to use hidden layers.

In order to add hidden layers, we need to answer these following two questions:

1. What is the required number of hidden layers?
2. What is the number of the hidden neurons across each hidden layer?

Following the previous procedure, the first step is to draw the decision boundary that splits the two classes. There is more than one possible decision boundary that splits the data correctly as shown in figure. The one we will use for further discussion is in figure 2(a).



a **b**

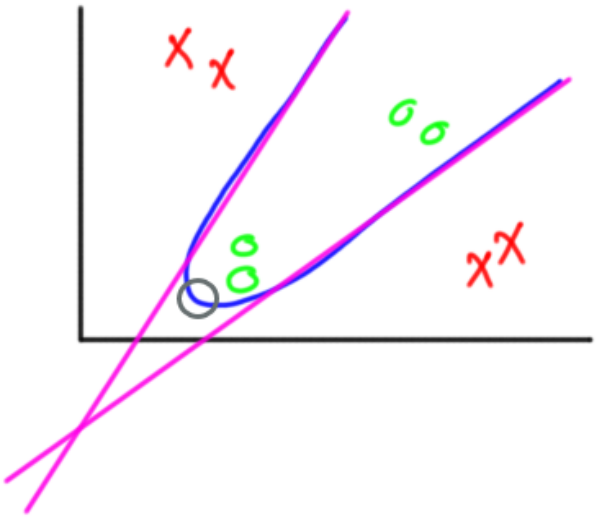
Following the guidelines, next step is to express the decision boundary by a set of lines.
The idea of representing the decision boundary using a set of lines comes from the fact that any ANN is built using the single layer perceptron as a building block. The single layer perceptron is a linear classifier which separates the classes using a line created according to the following equation:

$y = w_1*x_1 + w_2*x_2 + \cdots + w_i*x_i + b$

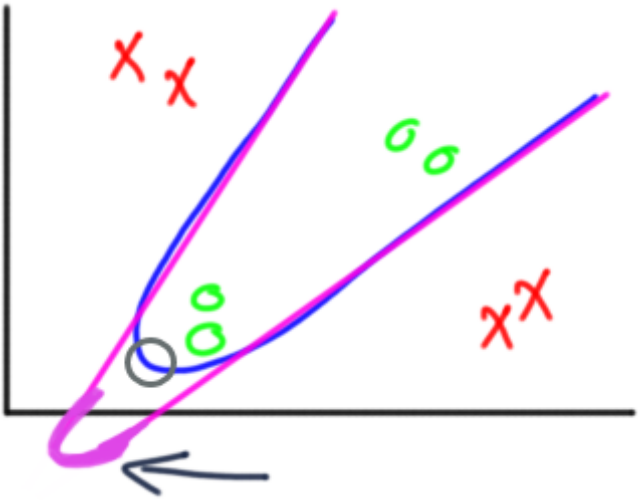
Where **x_i** is the input, **w_i** is its weight, **b** is the bias, and **y** is the output. Because each hidden neuron added will increase the number of weights, thus it is recommended to use the least number of hidden neurons that accomplish the task. Using more hidden neurons than required will add more complexity. Returning back to our example, saying that the ANN is built using multiple perceptron networks is identical to saying that the network is built using multiple lines.

In this example, the decision boundary is replaced by a set of lines. The lines start from the points at which the boundary curve changes direction. At such point, two lines are placed, each in a different direction.

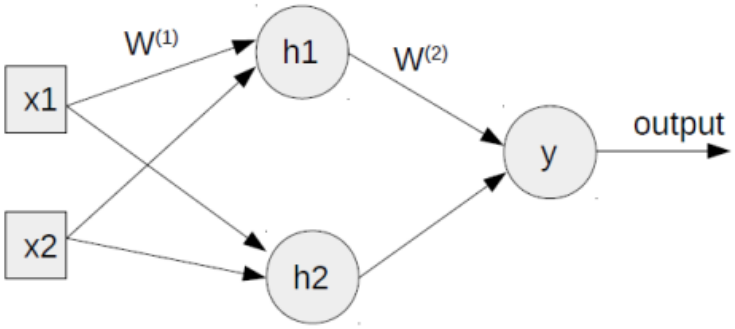
Because there is just one point at which the boundary curve changes direction as shown in figure by a gray circle, then there will be just two lines required. In other words, there are two single layer perceptron networks. Each perceptron produces a line.



Knowing that there are just two lines required to represent the decision boundary tells us that the first hidden layer will have two hidden neurons. Up to this point, we have a single hidden layer with two hidden neurons. Each hidden neuron could be regarded as a linear classifier that is represented as a line as in figure 3. There will be two outputs, one from each classifier (i.e. hidden neuron). But we are to build a single classifier with one output representing the class label, not two classifiers. As a result, the outputs of the two hidden neurons are to be merged into a single output. In other words, the two lines are to be connected by another neuron. The result is shown in figure.



Fortunately, we are not required to add another hidden layer with a single neuron to do that job. The output layer neuron will do the task. Such neuron will merge the two lines generated previously so that there is only one output from the network. After knowing the number of hidden layers and their neurons, the network architecture is now complete as shown in figure



Multilayer Perceptron in Python

```
from sklearn.neural_network import MLPClassifier
```

```
mlp = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(3, 3), random_state=1)

mlp.fit(X_train, y_train)

y_pred = mlp.predict(X_test)

accuracy = mlp.score(X_test, y_test)
```

Import the MLPClassifier class from the neural_network module of the scikit-learn library in Python. MLPClassifier stands for Multi-Layer Perceptron Classifier, which is a type of artificial neural network that can be used for classification tasks.

Create an instance of the MLPClassifier class with the following parameters:

- **solver='lbfgs'**: This parameter specifies the optimization algorithm used to train the neural network. 'lbfgs' is a quasi-Newton method that is often used for small datasets.

- **alpha=1e-5**: This parameter specifies the regularization strength for the neural network. A smaller value of alpha means less regularization, which can lead to overfitting.
- **hidden_layer_sizes=(3, 3)**: This parameter specifies the number of neurons in each hidden layer of the neural network. In this case, there are two hidden layers, each with three neurons.
- **random_state=1**: This parameter specifies the random seed used to initialize the weights of the neural network. Setting the random state to a fixed value ensures that the neural network will be initialized with the same weights every time the code is run, which can be helpful for reproducibility.

Lab Task

Predicting Who Survived the Titanic Using Multilayer Perceptron (MLP)

Goal

The goal of this lab task is to build a multilayer perceptron (MLP) classifier that can predict whether a passenger survived the sinking of the Titanic or not.

Dataset

The dataset contains information about the passengers of the Titanic, including features such as passenger ID, ticket class, name, age, gender, number of siblings and spouses, number of parents and children, ticket number, fare, cabin number, and port of embarkation. The target variable is whether the passenger survived the sinking of the Titanic (1 = Yes, 0 = No).

1. Import the necessary Python libraries, such as **pandas**, **numpy**, and **sklearn**.
2. Load the Titanic dataset into a pandas DataFrame using **pandas.read_csv()**.
3. Preprocess the data by converting categorical features into numerical ones, filling in missing values, and scaling the numerical features using **sklearn.preprocessing**.
4. Split the dataset into training and test sets using **sklearn.model_selection.train_test_split()**.
5. Build an MLP classifier using **sklearn.neural_network.MLPClassifier()** and train it on the training data.
6. Evaluate the performance of the MLP classifier on the test data using metrics such as accuracy, precision, recall, and F1-score.

7. Fine-tune the MLP classifier by adjusting its hyperparameters, such as the number of hidden layers, and the number of neurons per layer.
8. Evaluate the performance of the fine-tuned MLP classifier on the test data and compare it to the initial model.
9. Discuss the results and insights gained from the experiment, and identify potential areas for further improvement.